GraphLedger: Repository-Level Vulnerability Detection via Graph-Based Compression and Assumption Validation

Thi-Hong-Cuc Le
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology (HCMUT)
Ho Chi Minh City, Vietnam
Vietnam National University Ho Chi Minh City
Ho Chi Minh City, Vietnam
lthcuc.sdh242@hcmut.edu.vn

Hoang-Quoc-Bao Hua
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology (HCMUT)
Ho Chi Minh City, Vietnam
Vietnam National University Ho Chi Minh City
Ho Chi Minh City, Vietnam
hhqbao.sdh241@hcmut.edu.vn

Xuan-Bach Le*
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology (HCMUT)
Ho Chi Minh City, Vietnam
Vietnam National University Ho Chi Minh City
Ho Chi Minh City, Vietnam
lexuanbach@hcmut.edu.vn

Abstract

Repository-level vulnerability detection requires reasoning across files, procedures, and long-range dependency chains. This is difficult for both dominant families of methods: static analysis often over-approximates and produces many false positives, while LLM-based approaches struggle with large repository context and may generate plausible but unsupported claims.

We present **GraphLedger**, a framework that combines *Adaptive Assumption-Corroborating Compression* (AACC) with an *Assumption Ledger* for structured hypothesis verification. AACC reduces repository context while preserving vulnerability-relevant dependency structure. The LLM then proposes explicit vulnerability hypotheses over the compressed context, and the ledger validates their assumptions against repository evidence, producing *Verified*, *Rejected*, or *Inconclusive* outcomes. This makes repository-scale reasoning more auditable and less prone to unsupported conclusions.

GraphLedger targets taint-style, source–sink vulnerability hypotheses with bounded inter-procedural evidence, rather than arbitrary vulnerability classes. We evaluate it on CWE-Bench-Java [11, 14] and CVEfixes [1]. Across both datasets, AACC achieves 0.71 token reduction while retaining 0.90 of dependency edges and 0.88 of taint paths. On a held-out 100-repository CWE-Bench-Java test split, GraphLedger achieves 0.77 precision, 0.76 recall, and 0.76 F1, and reduces false discovery rate from 0.41 to 0.23 relative to VulAgent under a controlled shared-backend evaluation (§5.2). These results show that combining structure-aware compression with assumption validation improves the reliability and transparency of repository-level vulnerability detection.

CCS Concepts

• **Software and its engineering** → **Software verification and validation**; *Static analysis*; • **Security and privacy** → *Software security engineering*.

*Corresponding author.

Conference'17, Washington, DC, USA
2026. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords

Vulnerability detection, Code property graphs, Static analysis, Large language models, Repository analysis, Context compression

1 Introduction

Repository-level vulnerability detection requires reasoning across files, procedures, and long-range dependency chains. In realistic repositories, the source of attacker-controlled data, the transformations that preserve or block that influence, and the security-sensitive sink may appear in different files and procedures. This makes repository-scale analysis difficult for both dominant families of methods. Traditional static analysis provides broad coverage, but its conservative approximations often retain infeasible paths and therefore generate many false positives [7, 19]. LLM-based approaches offer stronger semantic reasoning, but repository context is often too large to fit naturally into a prompt, and partial context can lead to unstable cross-function conclusions or unsupported claims that the system never verifies [10, 14, 23].

We address this problem with **GraphLedger**, which formulates repository-level vulnerability detection as *structured hypothesis verification* over dependency graphs. GraphLedger combines *Adaptive Assumption-Corroborating Compression (AACC)*, which reduces repository context while preserving vulnerability-relevant reachability, with an *Assumption Ledger*, which checks hypothesis assumptions against explicit repository evidence. The ledger returns one of three outcomes: *Verified*, *Rejected*, or *Inconclusive*; once evidence contradicts an assumption, the corresponding hypothesis cannot later return to *Verified*. The pipeline is instantiated separately for each requested vulnerability class, using type-specific source–sink and evidence definitions.

Figure 1 summarizes the workflow. GraphLedger builds a code property graph, compresses it from G to G' with AACC, generates candidate vulnerabilities over the compressed graph, and then verifies the assumptions required by each hypothesis before returning a final decision. The workflow is executed independently for each vulnerability type rather than as a joint multi-vulnerability classifier. Static analysis establishes source–sink paths and dependency relations; the LLM then articulates explicit vulnerability hypotheses, and the ledger checks their required assumptions deterministically. The result is an auditable pipeline in which both retained context

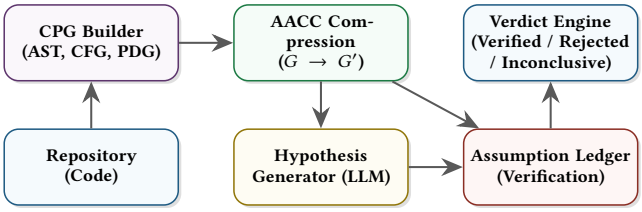


Figure 1: GraphLedger framework: repository-level vulnerability detection via compression, hypothesis generation, and assumption-aware verification.

and accepted claims are explicitly constrained, in line with the broader shift from black-box to glass-box, explainable AI-based software systems [13].

We evaluate GraphLedger on CWE-Bench-Java [11, 14] and CVE-fixes [1], combining a validated benchmark with repository-scale evidence. It reduces false discoveries while maintaining competitive recall under a shared-backend protocol isolating compression and verification effects (§5.2).

This paper makes two contributions:

- We present GraphLedger, a system co-designing vulnerability-aware compression (AACC) and monotonic assumption validation (Assumption Ledger) in a unified pipeline for taint-style, source–sink vulnerability detection at repository scale, positioned relative to CPG-guided LLM reasoning and agentic validators rather than general repository-level detection. The model captures bounded verification, conditional graph-preserving compression, and monotonic invalidation (Theorem 1). Unlike prior slicing, AACC's predicates are designed to retain nodes and edges required by the ledger's queries, so that evidence needed for bounded ledger checks is retained conditional on the compression assumptions of §3.
- We evaluate GraphLedger on CWE-Bench-Java [11, 14] and CVE-fixes [1]. AACC achieves 0.71 token reduction while preserving 0.90 dependency structure and 0.88 taint paths. On a held-out 100-repository CWE-Bench-Java test split, GraphLedger attains 0.77 precision, 0.76 recall, and 0.76 F1, reducing FDR from 0.41 to 0.23 vs. VulAgent [23] under a shared-backend protocol (§5.2). McNemar tests confirm significance (p=0.0081 vs. Neither on the test holdout).

The rest of the paper is organized as follows. Section 2 introduces the motivating example, Sections 3 and 4 present the formal model and pipeline, Sections 5 and 6 describe the experimental setup and results, Section 7 discusses implications and limitations, Section 8 reviews related work, and Section 9 concludes.

2 Motivating Example

Figure 2 illustrates the kind of repository-level SQL injection scenario that motivated **GraphLedger**. A request parameter is received in `UserController.java`, incorporated into a query in `QueryBuilder.java`, and executed in `Database.java`. A human reviewer can usually see why this path looks suspicious: the data flow crosses module boundaries, the query is assembled by string concatenation, and the sink eventually executes raw SQL. For an automated analyzer, however, turning that intuition into a justified report requires inter-procedural reasoning and explicit confirmation that no sanitization breaks the path.

The example surfaces two recurring difficulties in repository-scale detection: *context explosion* and *implicit assumptions*. Passing every transitively related file to the analyzer introduces enough peripheral code to obscure the relevant dependency path. At the same time, a claim such as "this path is vulnerable to SQL injection" holds only if input is user-controlled, sanitization is absent, and the sink executes non-parameterized SQL. In practice, many false alarms arise because such conditions are assumed rather than checked.

GraphLedger addresses both difficulties in sequence. AACC first constructs a code property graph and extracts a vulnerability-relevant slice that preserves the dependency chain from `id` to the database call while removing unrelated code. The LLM then proposes a concrete hypothesis over that compressed context: unsanitized user input may be concatenated into a query that later reaches execution. The Assumption Ledger turns this into an auditable verification process by decomposing the claim into explicit assumptions and retrieving repository evidence for each one. In Figure 2, data-flow analysis shows that `id` is user-controlled, code inspection shows that no sanitization occurs on the relevant path, and API evidence shows execution through `createStatement()` rather than a parameterized interface. The implementation applies this workflow independently to each requested vulnerability class; here the compression and validation steps are instantiated for SQL injection using type-specific source–sink and evidence definitions.

The example captures the central argument of the paper: repository-level vulnerability detection requires both selective compression and disciplined validation. Compression without verification leaves claims unchecked, while validation without compression is difficult to scale over repository context. GraphLedger combines the two in a single workflow, yielding more reliable detection and more transparent triage decisions than either stage alone [14, 16, 23].

3 Formal Model

We model repository-level vulnerability detection as *structured hypothesis verification* over inter-procedural graphs, following GraphLedger's stages: representation, hypothesis formation, conservative compression, and evidence-driven validation.

3.1 Repository Model

A repository is represented as a heterogeneous graph

$$G = (V, E_{AST}, E_{CFG}, E_{PDG}, E_{call}),$$

where nodes are program entities and edges capture syntax, control flow, dependence, and calls. Together, these relations form a Code Property Graph (CPG).

Table 1 summarizes the conservative assumptions used during graph construction. Under these assumptions, reachability

$$s \rightsquigarrow_G t$$

over-approximates feasible executions: any feasible dependency from s to t is represented by at least one path in G. The graph may include infeasible paths, but it does not omit relevant dependency structure. The "Feature" column lists common sources of

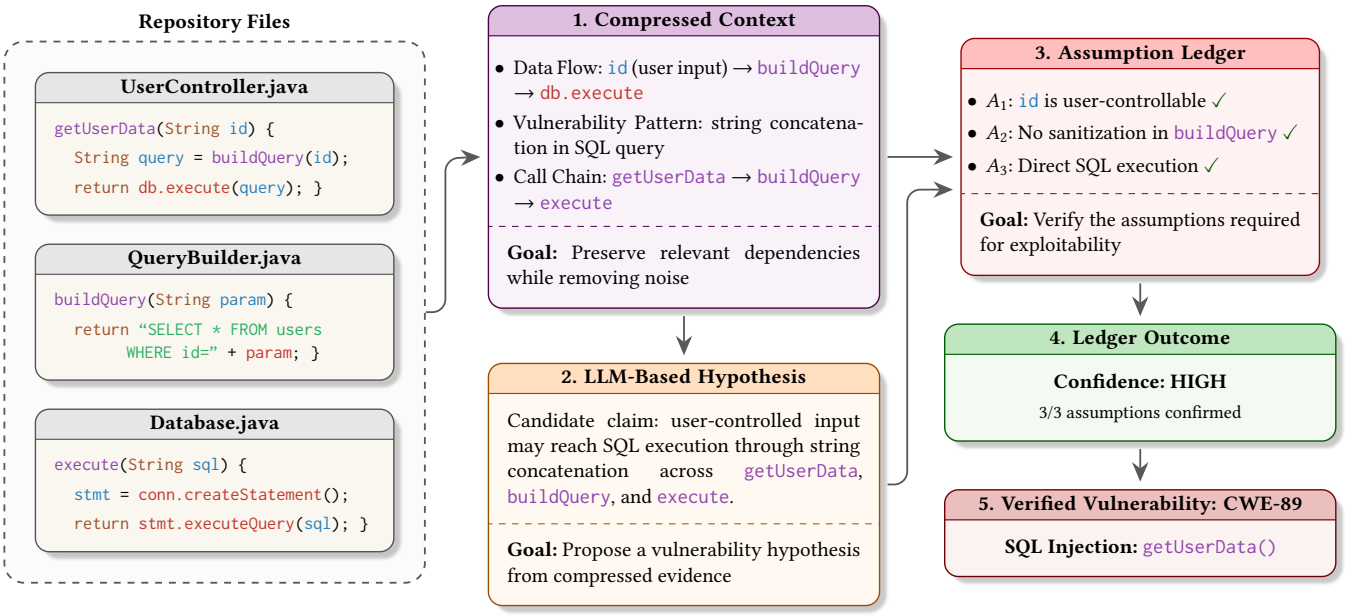


Figure 2: Repository-level SQL injection example. GraphLedger compresses inter-procedural context via AACC while preserving dependency structure, and validates hypotheses using the Assumption Ledger.

Table 1: Static-analysis assumptions in CPG construction.

Feature	Handling
Dynamic dispatch	Class Hierarchy Analysis (CHA) over-approximation
Aliasing	Flow-insensitive points-to analysis
Framework callbacks	Conservative entry modeling
Reflection / dynamic loading	Fallback dependency edges

ambiguity in repository-scale dependency recovery, and the "Handling" column shows how CPG construction resolves them conservatively. These assumptions define the static-analysis substrate for GraphLedger: AACC, witness-path reasoning, and ledger validation all rely on the CPG preserving source–sink reachability even when it includes infeasible paths.

3.2 Vulnerability Hypothesis

Let $\Pi_G(s,t)$ denote the set of dependency paths from a source s to a sink t. The implementation evaluates only a bounded subset:

$$\Pi_G^{(\ell,m)}(s,t) \subseteq \Pi_G(s,t),$$

where ℓ is the maximum path length and m is the maximum number of paths explored per source–sink pair. These bounds cap verification cost and stabilize behavior across repositories. In the experiments reported in this paper, we use $\ell = 20$ and $m = 8$, selected together with the *balanced* evidence preset via validation-only sensitivity (§5.1, supplementary Appendix H).

A hypothesis is

$$h = (s, t, \phi, \mathcal{A}),$$

where ϕ checks structural feasibility and $\mathcal{A} = \{A_1, \ldots, A_k\}$ is a set of semantic assumptions, such as input controllability or the absence of sanitization. Each A_i is a Boolean predicate over dependency paths. A path π is a *witness* if both structural feasibility and all assumptions hold:

$$\phi(\pi) \wedge \bigwedge_i A_i(\pi).$$

A hypothesis is satisfied if at least one witness path exists within the explored budget:

$$G \models_{\ell,m} h \triangleq \exists \pi \in \Pi_G^{(\ell,m)}(s,t) \,.\, \phi(\pi) \wedge \bigwedge_i A_i(\pi).$$

This bounded semantics matches the implementation rather than an idealized unbounded search.

3.3 Graph Compression

We model compression with an operator

$$C : \mathcal{G} \to \mathcal{G}, \qquad G' = C(G),$$

which reduces graph size while preserving vulnerability-relevant structure. Compression is *path-sound* if every path in the compressed graph corresponds to some path in the original graph:

$$\forall \pi' \in \Pi_{G'}(s,t),\ \exists \pi \in \Pi_G(s,t) \text{ s.t. } \pi' \preceq \pi,$$

where $\pi' \preceq \pi$ means π' is an abstraction of π.

Define the set of vulnerability-reachable pairs as

$$Paths_{vuln}(G) = \{(s,t) \mid \exists \pi \in \Pi_G(s,t),\ \phi(\pi)\}.$$

Compression is conservative if

$$Paths_{vuln}(G') \subseteq Paths_{vuln}(G).$$

This rules out compression-induced structural false positives, although some true paths may still be removed.

When compression uses behavioral summaries, we treat each summary edge $u \rightsquigarrow v$ as an admissible abstraction only if there exists a concrete dependency path $\pi \in \Pi_G(u, v)$ inside the summarized procedure or callee region. Summaries may hide internal nodes, but they may not introduce cross-boundary dependencies without such a path. Under this path-backed interpretation, every abstract path in G' still corresponds to a concrete path in G, possibly with summarized segments expanded to their concrete realizations.

3.4 Assumption Ledger

The Assumption Ledger tracks the status of each assumption as new evidence is collected. Each assumption is in one of three states:

- *pending*: not yet resolved,
- *validated*: supported by evidence,
- *undermined*: contradicted by evidence.

Let

$$State : \mathcal{A} \to \{pending, validated, undermined\}$$

denote the current state mapping. When new evidence arrives, an assumption may move from *pending* to either *validated* or *undermined*; otherwise its state remains unchanged.

A hypothesis is ledger-valid iff no assumption is contradicted:

$$valid_L(h) \triangleq \forall A_i \in \mathcal{A},\ State(A_i) \neq undermined.$$

We also write

$$complete_L(h) \triangleq \forall A_i \in \mathcal{A},\ State(A_i) = validated,$$

meaning that every assumption in h has been validated.

The final verdict is

$$Verdict(h, L) = \begin{cases} Rejected & \text{if } \neg valid_L(h), \\ Verified & \text{if } G \models_{\ell,m} h \ \wedge\ complete_L(h), \\ Inconclusive & \text{otherwise.} \end{cases}$$

Inconclusive denotes cases where the hypothesis is neither confirmed nor rejected.

3.5 Verification and Monotonicity

A hypothesis is verified only when the explored graph provides structural support and the ledger validates all assumptions:

$$\text{Verified}_G(h, L) \triangleq \left(\exists \pi \in \Pi_G^{(\ell,m)}(s, t)\ .\ \phi(\pi) \wedge \bigwedge_i A_i(\pi)\right) \wedge \left(\forall A_i \in \mathcal{A}\ .\ State(A_i) = validated\right).$$

We assume monotonicity: once an assumption becomes *undermined*, it is not reverted. This prevents oscillatory decisions and guarantees convergence to a stable outcome after finitely many assumption updates.

Theorem 1 (Monotonic Rejection Under Bounded Verification). *If any assumption in $\mathcal{A}$ is undermined, the hypothesis verdict cannot later become Verified; it remains Rejected in all later steps.*

Supplementary Appendix E gives the full inductive proof.

Algorithm 1 Adaptive Assumption-Corroborating Compression

Require: Repository graph G
Ensure: Compressed graph G'

```
CPG ← BuildCPG(G)
S ← ComputeDependencySlices(CPG)
CPG' ← PruneNonPreserving(CPG, S)
B ← GenerateBehaviorSummaries(CPG')        ▷ optional
G' ← AssembleCompressedGraph(CPG', B)
return G'
```

3.6 Scope of Formal Guarantees

The formal properties differ in strength. Theorem 1 (monotonic rejection) is unconditional: once an assumption is undermined, the hypothesis cannot return to *Verified*. Proposition 1 (compression safety) is conditional on witness-backed behavioral summaries, empirically supported (*TaintPres* = 0.88, Section 6.2) but not formally verified. Behavioral summary extraction, CHA call-graph over-approximation, and LLM hypothesis generation are heuristic components with no formal guarantees; the ledger mitigates LLM fallibility through deterministic evidence checks.

4 Methodology

This section instantiates the formal model of Section 3 as an operational pipeline with three stages: compression, hypothesis validation, and verdict reporting.

4.1 AACC

AACC maps a repository dependency graph G to a reduced graph G' by removing irrelevant regions while preserving the dependency structure needed for source–sink reasoning.

Graph Construction. We construct CPGs using Joern for Java and a Tree-sitter–based extractor for Python, with interprocedural dataflow recovery in both settings and CHA fallback edges in the call graph. Algorithm 1 then computes dependency slices, prunes nonpreserving nodes, and optionally attaches behavioral summaries.

Supported Vulnerability Types. The pipeline is instantiated per taint-oriented vulnerability class, each with its own source–sink definitions and evidence rules. Target types are *user-specified* (via the CLI or the benchmark's CWE-to-type mapping), not inferred from the code. Thus AACC is run separately for each requested type, producing type-specific slices and compressed graphs. Supplementary Appendix A lists the supported classes and representative CWE mappings.

PruneNonPreserving. Algorithm 2 removes nodes outside all retained dependency slices and source–sink corridors, except when needed for alias-mediated dependencies or summary references. AliasKeep preserves nodes whose removal would break value-flow through aliases, while SummaryRef preserves nodes referenced by an attached behavioral summary.

Connecting Implementation to Formalism. Algorithm 1 implements S_{src} and S_{sink} via CWE-specific marker patterns (supplementary Appendix A). PruneNonPreserving (Algorithm 2) computes the reachability closure $R_s \cap R_t$ from Section 3. AliasKeep and

SummaryRef extend the corridor with nodes the ledger may query; AliasKeep is a lightweight name/signature heuristic, not a full points-to analysis.

Behavioral Summaries. When enabled, AACC attaches lightweight behavioral summaries to retained procedures via a *static AST-scanning pass* (not the LLM) that derives coarse read/write sets and security tags (sanitization, authorization, SQL execution, path normalization, output encoding). Summaries are attached as abstract procedure nodes, preserving high-level security cues without reintroducing full callee structure.

Complexity. AACC's cost is $O(t \cdot (|V|+|E|))$ for slicing, where t is the number of requested vulnerability types, dominated by source–sink reachability. For the evaluated repositories (average 300K LOC, average CPG ~10K nodes), per-type compression completes in under 2 seconds. See supplementary Appendix D for the detailed analysis.

Summary Safety Property. Summary edges are introduced only for methods already retained in the compressed subgraph, with optional witness metadata linking them to concrete paths in G. The witness-backing condition is not mechanized but is validated empirically (TaintPres = 0.88, §6.2).

Proposition 1 (Compression Safety — Conditional). *The compressed graph G' is safe for bounded verification if:*

(1) PruneNonPreserving removes only nodes outside the source–sink dependency closure induced by $\mathcal{S}$ and $R_s \cap R_t$.
(2) Behavioral summaries materialize only witness-backed abstract edges whose endpoints are connected by concrete paths in G.
(3) Bounded verification follows Section 3.

Formally:

$$\exists \pi' \in \Pi_{G'}^{(\ell,m)}(s,t) \Rightarrow \exists \pi \in \Pi_G(s,t).$$

Assumptions (1) and (3) are enforced by construction. Assumption (2) is an engineering invariant, supported empirically by TaintPres = 0.88 (Section 6.2).

Supplementary Appendix E gives the full proof.

Hypothesis Prompting. GraphLedger uses the LLM as a bounded hypothesis generator over G'. The prompt provides a CPG-derived summary and target vulnerability type, and requests a structured JSON object containing a concrete source-to-sink path grounded in visible CPG entities and an ordered list of falsifiable assumptions (e.g., "no input sanitization between `getParameter()` and `executeQuery()`"). The output is a structured claim, not a final verdict. Supplementary Appendix C gives the full prompt templates.

4.2 Ledger-Based Hypothesis Validation

Algorithm 3 updates the evidential status of each assumption attached to a hypothesis. Repository evidence may support, contradict, or leave unresolved an assumption. A hypothesis is rejected only when a required assumption is explicitly undermined; otherwise it remains *Verified* or *Inconclusive*. For an assumption A_i, evidence e is interpreted as validated, undermining, or inconclusive.

State transitions depend on an evidence interpretation function:

$$\mathcal{E}(A_i, e) \in \{\textit{validated}, \textit{undermined}, \textit{inconclusive}\}.$$

Algorithm 3 Hypothesis Validation Using the Assumption Ledger

Require: Hypotheses $\mathcal{H}$ over graph G'
Ensure: Verdict mapping $\mathcal{V}$ over $\mathcal{H}$

$\mathcal{L} \leftarrow \emptyset$
for all $h \in \mathcal{H}$ **do**
  RegisterHypothesis($\mathcal{L}, h$)
end for
for all assumption $A_i \in \mathcal{L}$ **do**
  $e \leftarrow$ FetchEvidence(G', A_i)
  UpdateState($\mathcal{L}, A_i, e$)
  if $State(A_i) = \textit{undermined}$ **then**
    InvalidateDependents($\mathcal{L}, A_i$)
  end if
end for
$\mathcal{V} \leftarrow$ AssignVerdicts($\mathcal{L}, G'$)
return $\mathcal{V}$

Validated and undermining evidence update the ledger state directly, while inconclusive evidence leaves the assumption *pending*; unresolved assumptions yield an *Inconclusive* hypothesis verdict.

Evidence Classification Rules. The interpretation function $\mathcal{E}$ is shared across all CWE types and maps each assumption description to one of four keyword-based categories, then combines that category with retrieved CPG evidence:

Table 2: Evidence interpretation rules for $\mathcal{E}(A_i, e)$: rows are assumption categories, columns are evidence types, and cells are resulting states.

Assumption class	Sanitizer	Flow	None
Absence ("no sanitization")	undermined	validated	pending
Presence ("sanitizer exists")	validated	undermined	pending
Flow ("taint reaches sink")	—	validated	pending
Default	undermined	validated	pending

For example, "no input sanitization between source and sink" is an *Absence* assumption: a reachable sanitizer marker undermines it, confirmed flow without such a marker validates it, and insufficient evidence leaves it pending. Sanitizer markers are CWE-specific; full marker lists are included in the artifact.

Evidence Retrieval. Evidence is collected through bounded dependency traversal, optionally augmented with semantic matching over behavioral summaries. Candidate nodes are generated structurally, ranked by relevance, and then classified by the rule set above. Two hyperparameters control the precision–recall trade-off: the maximum dependency depth and the minimum summary-similarity threshold. These are tuned on validation repositories and then fixed for test-time evaluation. For auditability, the artifact logs retrieved candidates, traversal depth, similarity scores, and final evidence classifications.

Evidence retrieval is intentionally *conservative*: assumptions are invalidated only by explicit contradicting evidence. Under this policy, rejection is sound, while unresolved assumptions remain *Inconclusive* rather than being forced to *validated*:

$$\mathcal{E}(A_i, e) = \textit{undermined} \Rightarrow \textit{Rejected}(h).$$

Algorithm 2 PruneNonPreserving($CPG, \mathcal{S}$)

Require: $CPG = (V, E)$, dependency slices $\mathcal{S}$, sources $\mathcal{S}_{src}$, sinks $\mathcal{S}_{sink}$
Ensure: Pruned graph CPG'
$K \leftarrow \bigcup_{S \in \mathcal{S}} S$ ▷ slice nodes to keep
$R_s \leftarrow$ ForwardReach($CPG, \mathcal{S}_{src}$)
$R_t \leftarrow$ BackwardReach($CPG, \mathcal{S}_{sink}$)
for all $v \in V$ **do**
  $keep(v) \leftarrow (v \in K) \vee (v \in R_s \cap R_t) \vee$ AliasKeep(v) $\vee$ SummaryRef(v)
end for
$V' \leftarrow \{v \in V \mid keep(v)\}$
$E' \leftarrow \{(u, w) \in E \mid u \in V' \wedge w \in V'\}$
return $CPG' = (V', E')$

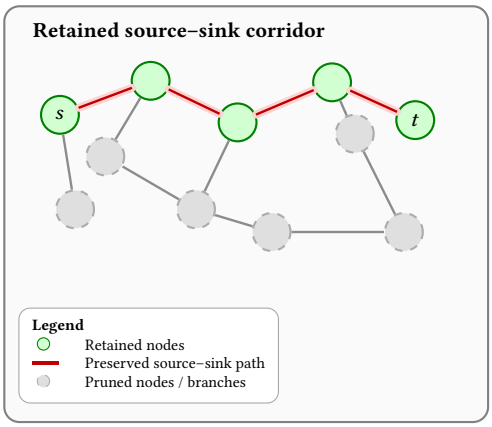


Figure 3: Intuition for PruneNonPreserving: pruning removes nodes outside source–sink dependency closures while preserving vulnerability-relevant reachability.

Table 3: Sensitivity of evidence retrieval heuristics, computed over the full 120-repository universe (not the test holdout). The validation-only version used for preset selection is in supplementary Appendix H.

Setting	Precision↑	Recall↑	FDR↓
Low (h=5, τ=0.0, any)	0.73	0.57	0.27
Balanced (h=10, τ=0.3, any)	0.74	0.70	0.26
High (h=20, τ=0.5, all)	0.68	0.79	0.32

Table 3 varies maximum dependency distance h, similarity threshold τ, and combine mode ('any' = structural or similarity match; 'all' = both). The stricter low setting improves precision but lowers recall, while the more permissive high setting raises recall at the cost of higher FDR. We therefore use the balanced configuration, which gives the best validation trade-off.

5 Experimental Setup

This section describes the experimental setup used to assess semantic preservation, detection reliability, and scalability under controlled, reproducible conditions.

5.1 Datasets

We use two evaluation datasets, CWE-Bench-Java [11, 14] and CVEfixes [1]. Together, the two datasets provide broad repository diversity together with a clean benchmark for evaluation.

CVEfixes [1] is a large-scale dataset that automatically links vulnerability records from the National Vulnerability Database (NVD) to fixing commits in open-source repositories. It provides paired vulnerable and patched code, together with metadata such as CVE descriptions, CWE labels, and commit information. This makes it useful for repository-scale evaluation.

CWE-Bench-Java [11, 14, 15] is a manually validated dataset of vulnerabilities from real-world Java projects, reducing the label noise and unrelated code changes common in automatically mined datasets. It emphasizes interprocedural, cross-module security flaws that are difficult to capture with purely local analysis [22, 29]. The dataset contains 120 repositories with an average size of 300K LOC and covers four CWE classes: path traversal (CWE-22, 55 repos), cross-site scripting (CWE-79, 31 repos), code injection (CWE-94, 21 repos), and command injection (CWE-78, 13 repos) [14].

To prevent data leakage—here meaning near-duplicate code shared across training and evaluation corpora that can inflate scores, not evaluation-label leakage—we apply strict deduplication before evaluation [14, 24], removing exact and near-duplicate clones at both the repository (source) and intermediate-representation levels.

Validation and Test Splits. For CWE-Bench-Java, we reserve 20 repositories (out of 120) as a validation holdout used *only* for selecting evidence-retrieval hyperparameters (h=10, τ=0.3, combine=any) and bounded verification parameters (ℓ=20, m=8) by maximizing F1; these were then frozen for all subsequent evaluations. **Primary detection metrics** (Table 5) are computed on the remaining **100 test repositories** (n=200 paired instances). Repositories are split at `project_slug` granularity, so no repository appears in both sets; split membership is frozen in the artifact (`cwe_validation_slugs.txt`, `cwe_test_slugs.txt`, seed 42). Supplementary Appendix G retains the all-120 results from the submitted version, and supplementary Appendix I reports robustness over 50 random 20/100 holdouts.

False Discovery Rate (FDR).. Throughout the paper we report FDR = FP/(TP + FP), the fraction of positive predictions that are false positives, as a proxy for analyst triage burden.

CVEfixes Protocol (separate). CVEfixes detection uses an independent protocol: n=100 Java projects sampled with seed 42 from the processed corpus (per-case manifest in the artifact). This sample is *not* drawn from the CWE-Bench-Java test holdout, and the two benchmarks are never mixed.

Table 4: Evaluation baselines and integration kind. *Offline adapters* convert external tool outputs into our result schema; *inline reimplementations* are controlled re-implementations sharing checkout, CPG, and LLM backend where applicable; the gpt-4o-mini row is an *ablated GraphLedger* (compression and ledger disabled), not a standalone third-party tool.

Baseline	Category	Kind
CodeQL [7]	Static Analysis	Offline adapter
IRIS [16]	Specification Inference	Offline adapter
LLMAO [27]	LLM Fault Localization	Offline adapter
RepoAudit [8]	Repository Auditing Agent	Offline adapter
LLMxCPG [14]	CPG + LLM Reasoning	Offline adapter
GraphRAG [4]	Graph Retrieval Reasoning	Inline reimpl.
SemTaint [6]	Agentic Repair	Inline reimpl.
Vul-RAG [3]	Retrieval-Based Analysis	Inline reimpl.
VulAgent [23]	Agentic Detection	Inline reimpl.
gpt-4o-mini [20]	Single-Agent LLM	Ablated GraphLedger

Evaluation limitation. The submitted version evaluated all 120 repositories, including the 20 used for hyperparameter selection; the exact validation split was not archived. This camera-ready reports primary metrics on a documented 100-repository test holdout (seed 42). All-120 results are in supplementary Appendix G.

CVEfixes Label Quality. CVEfixes is automatically mined from NVD-to-commit links and contains label noise from unrelated code changes [1]. The higher invalidation rate (36% vs. 18% on CWE-Bench-Java) partly reflects this; the manually validated CWE-Bench-Java provides a cleaner reference point.

The combined dataset covers 90 CWE labels suited to source–sink reasoning across files and procedures. CWE-Bench-Java contributes path traversal (CWE-22), command injection (CWE-78), cross-site scripting (CWE-79), and code injection (CWE-94); CVEfixes adds SQL injection (CWE-89), unsafe deserialization (CWE-502), authorization bypass (CWE-862/863), and SSRF (CWE-918). At the file level, CVEfixes provides 1,198 processed instances with 3,449 vulnerable/fixed file pairs across 532 CVEs, while CWE-Bench-Java contributes 358 vulnerable files across 120 manually validated CVEs.

5.2 Baselines

We compare GraphLedger with representative methods spanning static analysis, graph-based reasoning, retrieval-augmented LLMs, and agent-based vulnerability analysis. Baseline execution is unified through a shared runner that converts all methods to a common detector-format output and evaluates them under matched repository inputs and model access when applicable.

Shared-Backend Protocol. **All LLM-based methods are evaluated with gpt-4o-mini as the shared backend[1] unless otherwise noted.** This protocol isolates differences in repository representation, context selection, prompting structure, and verification logic rather than differences caused by foundation-model scale. Our results support the claim that GraphLedger performs strongly under a controlled evaluation regime, not the stronger claim that it uniformly dominates all prior systems in their individually optimized settings. Systems such as LLMxCPG or VulAgent were originally studied with task-specific components, custom prompts, or fine-tuned backends; re-evaluating them under a shared LLM may understate their best achievable numbers.

[1]See §6.4 for comparison with DeepSeek and LLaMA 3.1.

Baseline Configuration. All baselines were evaluated under controlled settings; static analyzers used recommended rule sets, and LLM-based systems received matched repository inputs and evaluation protocols. Detailed per-method configurations, tuning choices, and prompt setups are listed in supplementary Appendix B.

As summarized in Table 4, some baselines are interfaced through offline adapters over externally generated outputs, while others are controlled inline reimplementations of the original method design. All baselines score against the same paired ground truth and evaluator, and known simplifications (e.g., the LLMAO score threshold) are documented in the artifact. We therefore interpret the baseline comparison as a controlled matched-protocol comparison, not as a claim that every baseline is outperformed in its native standalone configuration.

For GraphLedger, the evidence-retrieval thresholds were selected on the validation split and then frozen for the test split. To support reproducibility, the implementation includes the final dependency-distance and summary-similarity thresholds, together with per-assumption retrieval logs recording candidate evidence, ranking scores, and final ledger-state updates.

Ablation variants isolate the individual contributions of analysis-aware compression and assumption-ledger verification.

5.3 Statistical Evaluation Protocol

All experiments follow a controlled protocol to reduce stochastic variation in LLM-based systems. The implementation supports seeded execution together with bootstrap confidence intervals and paired significance tests for method comparison. Detection differences are evaluated with the **McNemar test**, and paired runtime comparisons with the **Wilcoxon signed-rank test**. Unless otherwise stated, bounded verification uses maximum path length 20 and at most 8 paths per source–sink pair.

Hardware and Environment. All experiments were conducted on three NVIDIA H100 Mini servers (20 GB VRAM each). Baseline evaluations were distributed across the three servers under identical software configurations (Ubuntu 22.04, Python 3.11, CUDA 12.2). LLM inference was performed via API (gpt-4o-mini) or local deployment (DeepSeek, LLaMA 3.1) on the same hardware.

Main-text tables report rounded summary values under this shared evaluation policy. Dataset splits, prompts, configurations, and evaluation scripts will be released to support reproducibility.

6 Results and Analysis

We evaluate GraphLedger along four dimensions: detection performance, compression quality, verification behavior, and efficiency.

6.1 RQ1: Vulnerability Detection Performance

Tables 5 and 6 report the matched-baseline detection comparison on CWE-Bench-Java and CVEfixes under the shared protocol of

Table 5: Detection performance on the CWE-Bench-Java 100-repository test holdout (n=200 paired instances). Hyperparameters were selected on the 20-repository validation holdout (§5.1); all-120 results are in supplementary Appendix G.

Method	Precision↑	Recall↑	F1↑	FDR↓
Static / Spec.-Based				
CodeQL [7]	0.75	0.46	0.57	0.25
IRIS [16]	0.58	0.60	0.59	0.42
LLM / Retrieval-Augmented				
gpt-4o-mini [20]	0.54	0.79	0.64	0.46
GraphRAG [4]	0.56	0.70	0.62	0.44
Vul-RAG [3]	0.53	0.71	0.61	0.47
LLM Defect Localization				
LLMAO [27]	0.64	0.67	0.66	0.36
Agentic Analysis				
RepoAudit [8]	0.66	0.73	0.70	0.34
SemTaint [6]	0.62	0.49	0.55	0.38
VulAgent [23]	0.59	0.70	0.64	0.41
Structure-Aware LLM				
LLMxCPG [14]	0.68	0.75	0.71	0.32
GraphLedger (Ours)	**0.77**	**0.76**	**0.76**	**0.23**

Table 6: Detection performance on CVEfixes (n=100). CVEfixes uses an independent seed-42 project sample, disjoint from the CWE-Bench-Java test holdout (§5.1).

Method	Precision↑	Recall↑	F1↑	FDR↓
Static / Spec.-Based				
CodeQL [7]	0.65	0.40	0.50	0.35
IRIS [16]	0.52	0.55	0.53	0.48
LLM / Retrieval-Augmented				
gpt-4o-mini [20]	0.50	0.63	0.56	0.50
GraphRAG [4]	0.56	0.65	0.60	0.44
Vul-RAG [3]	0.58	0.66	0.62	0.42
LLM Defect Localization				
LLMAO [27]	0.60	0.63	0.61	0.40
Agentic Analysis				
RepoAudit [8]	0.61	0.65	0.63	0.39
SemTaint [6]	0.59	0.67	0.63	0.41
VulAgent [23]	0.61	0.64	0.62	0.39
Structure-Aware LLM				
LLMxCPG [14]	0.64	0.67	0.65	0.36
GraphLedger (Ours)	**0.74**	**0.71**	**0.72**	**0.26**

Section 5. We emphasize FDR as a proxy for analyst triage burden:

$$\text{FDR} = FP/(TP + FP)$$

Unless noted otherwise, the reported results treat *Inconclusive* as negative; a triage-oriented alternative is discussed in Section 7.1. Readers seeking per-CWE breakdowns can refer to supplementary Appendix F.

The overall pattern is consistent across both datasets. Static/specification baselines keep FDR relatively low but miss repository-specific flows, whereas LLM-heavy baselines recover recall at the cost of more false positives. GraphLedger yields the best trade-off: on the CWE-Bench-Java test holdout it improves over VulAgent from 0.59 to 0.77 precision, from 0.70 to 0.76 recall, and from 0.41 down to 0.23 FDR, and over the strongest baseline (LLMxCPG, F1 0.71) by 5 F1 points; on CVEfixes it reaches 0.74 precision, 0.71 recall, 0.72 F1, and 0.26 FDR. The gain comes from filtering unsupported hypotheses before they become final findings.

Statistical Significance. On CWE-Bench-Java (n=200), McNemar tests show the full pipeline significantly outperforms Neither (p=0.0081) and No Ledger (p=0.0377); Full vs. No Compression is directionally consistent but not significant (p=0.2070). On CVEfixes (n=100), only Full vs. Neither is significant (p=0.0162); other comparisons are directionally consistent but underpowered. Bootstrap 95% accuracy CIs are [0.705, 0.825] (CWE-Bench-Java) and [0.650, 0.830] (CVEfixes).

Per-CWE-Class Performance. GraphLedger performs best on vulnerabilities with clear source–sink structure. On CWE-Bench-Java, path traversal and command injection are strongest; the `commandinjection` detector transfers to 15/21 CWE-94 cases (P = 0.83). On CVEfixes, authorization bypass and SQL injection show broader coverage. Supplementary Appendix F provides full per-CWE tables. This comparison is matched-backend to isolate compression and verification design.

6.2 RQ2: Compression Effectiveness

We next ask whether AACC reduces repository context without discarding reasoning-critical structure. Let $G = (V, E)$ and $G' = (V', E')$ denote the original and compressed graphs. Because the retained graph is serialized for downstream reasoning, we report token reduction rather than node reduction:

$$\text{TokenReduction} = 1 - T(G')/T(G).$$

Here, $T(G)$ and $T(G')$ denote the tokenized sizes of the original and compressed graph serializations.

We also report taint-path preservation,

$$\text{TaintPres}(G, G') = \frac{|\text{TaintPaths}(G') \cap \text{TaintPaths}(G)|}{|\text{TaintPaths}(G)|}.$$

dependency preservation,

$$\text{Dep}(G, G') = \frac{|\text{DepEdges}(G) \cap \text{DepEdges}(G')|}{|\text{DepEdges}(G)|}.$$

and context efficiency,

$$CE(G, G') = \frac{\text{TaintPres}(G, G')}{1 - \text{TokenReduction}}.$$

Here, TaintPaths$(\cdot)$ denotes vulnerability-relevant taint paths and DepEdges$(\cdot)$ dependency edges. Higher TaintPres indicates fewer vulnerability-relevant paths are lost during compression. Higher Dep indicates less structural context for dependency reasoning is discarded. Higher CE indicates more taint-relevant structure preserved per unit of context removed. CE is a derived ratio of two empirical quantities (preservation and token reduction); it

is a descriptive efficiency measure, not an additional soundness guarantee.

Table 7: Compression quality results. Unlike the detection tables, these compression statistics are computed over all evaluated cases (the full 120-repository universe), not the test holdout; they characterize AACC's behavior, not detection performance.

Method	TokenRed.↑	Dep.↑	TaintPres.↑	CE↑
Naive Truncation	0.62	0.54	0.49	0.31
GraphRAG [4]	0.58	0.71	0.68	0.49
LLMxCPG [14]	0.65	0.83	0.80	0.63
AACC (Ours)	**0.71**	**0.90**	**0.88**	**0.79**

Table 7 shows the strongest trade-off for AACC. *Naive Truncation* is a budget-matched lower bound that keeps CPG nodes in breadth-first order without vulnerability-aware preservation. AACC removes more context than Naive Truncation and GraphRAG while retaining more dependency and taint structure; it also improves over LLMxCPG, raising token reduction from 0.65 to 0.71 and CE from 0.63 to 0.79. In short, AACC removes mostly irrelevant code while empirically retaining most of the paths used by bounded ledger checks (TaintPres = 0.88).

6.3 RQ3: Hypothesis Verification Consistency

We analyze how often the ledger overturns candidate findings, finds supporting evidence, and leaves cases deferred under bounded search. We report invalidation rate and evidence coverage as

$$IR = |A_u|/|A|, \qquad EC = |A_e|/|A|.$$

We further report hypothesis stability and inconclusive verdict rate:

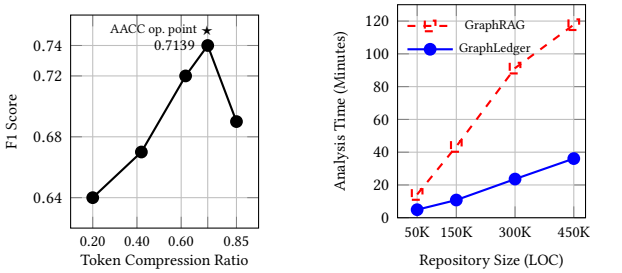
$$HS = \frac{1}{|H_v|} \sum_{h \in H_v} \text{stability}(h), \qquad IVR = \frac{\text{inconclusive hypotheses}}{\text{total hypotheses}}.$$

Here, A is the set of tracked assumptions, A_u the undermined assumptions, A_e the assumptions for which supporting or contradicting evidence is retrieved, and H_v the set of validated hypotheses. stability(h) denotes the empirical support score of a validated hypothesis as additional evidence is explored.

Table 8: Verification behavior over both datasets. These ledger-behavior statistics are computed over all evaluated cases (the full 120-repository universe for CWE-Bench-Java), not the test holdout; detection metrics on the holdout are in Table 5.

Metric	CVEfixes	CWE-Bench-Java	Overall
Invalidation Rate (IR)↓	36%	18%	28%
Evidence Coverage (EC)↑	82%	86%	84%
Hypothesis Stability (HS)↑	90%	89%	90%
Inconclusive Verdict Rate (IVR)↓	14%	8%	11%

Table 8 shows a clear dataset difference. On CVEfixes, the ledger invalidates more hypotheses (36% vs. 18%), suggesting noisier candidates. On CWE-Bench-Java it is more decisive, with a lower inconclusive rate (8% vs. 14%). Across both datasets, evidence coverage is high (84%) and hypothesis stability stays near 90%, suggesting that once relevant evidence is found, conclusions are usually stable.



(a) F1 as a function of token compression ratio. **(b) Runtime growth with repository size.**

Figure 4: RQ4 efficiency trends.

6.4 LLM Backend Sensitivity

To test backend dependence, we re-ran GraphLedger on a class-balanced CVEfixes subsample with gpt-4o-mini, DeepSeek, and LLaMA 3.1 under the balanced evidence-retrieval setting (h=10, τ=0.3, combine=any). Minor numerical differences from Table 6 reflect subsample composition.

Across the three backends the results stay in a narrow band — F1/FDR of 0.74/0.26 (gpt-4o-mini), 0.68/0.30 (DeepSeek), and 0.68/0.33 (LLaMA 3.1). gpt-4o-mini yields the best F1 and lowest FDR, but the differences are not statistically significant (McNemar: vs. DeepSeek p=0.2673; vs. LLaMA 3.1 p=0.0771). Since all three backends keep FDR below 0.33, most gains appear to come from the AACC+Ledger design rather than model-specific behavior; full per-backend numbers are in the artifact (`results_full/sensitivity/`).

6.5 RQ4: Component Efficiency

We next ask whether the two core components contribute independently and whether the full pipeline remains efficient at scale.

Component Contribution. We first vary compression strength under identical reasoning settings, then measure runtime growth with repository size.

Figure 4(a) shows an expected peak: F1 rises from 0.64 at 20% compression to 0.74 at the AACC operating point (71.39%), then drops when over-compression removes dependency-relevant paths. We then compare four variants: neither component, compression only, ledger only, and the full system.

Table 9: Component contribution (CWE-Bench-Java, 100-repository test holdout).

Configuration	Precision↑	Recall↑	F1↑	FDR↓
Neither	0.60	0.72	0.66	0.40
Compression	0.67	0.68	0.67	0.33
Ledger	0.70	0.73	0.71	0.30
Full	**0.77**	**0.76**	**0.76**	**0.23**

Ablation Results. Table 9 shows that both components help, but in different ways. The *Neither* setting still uses the full CPG, just without AACC or ledger validation, so it is the right no-component reference for GraphLedger rather than an LLM-only baseline. AACC mainly improves input quality, moving precision from 0.60 to 0.67 and FDR from 0.40 to 0.33; the ledger adds complementary false-positive control, reaching 0.70 precision and 0.30 FDR. The full system performs best at 0.77 precision and 0.23 FDR, showing that cleaner evidence and explicit validation are complementary.

Efficiency and Scalability. Under the bounded setting of Section 5 ($\ell = 20$, $m = 8$ per source–sink pair), Figure 4(b) shows near-linear runtime growth for GraphLedger: 4.9–36.1 minutes over the 50K–450K LOC range, with a median of 10.4 minutes. GraphRAG grows roughly 3–4× faster over the same range (14.2–117.8 minutes). The reported runtimes include behavioral summary computation, which adds about 15% to AACC time but provides security-tag evidence used by the ledger. Overall, the added reasoning structure remains practical on the evaluated repositories.

Prompt Sensitivity. We did not ablate prompt wording, so prompt sensitivity remains a limitation. The ledger provides a partial safeguard by validating hypotheses against repository evidence before they become final verdicts.

7 Discussion

The results support the paper's main claim: repository-scale vulnerability detection becomes more reliable when structure-aware compression is coupled with explicit hypothesis verification. In GraphLedger, AACC reduces context while preserving reasoning-critical relations, and the Assumption Ledger turns model outputs into auditable claims before acceptance. The clearest gains therefore appear in precision, false discovery rate, and verification transparency rather than in exhaustive vulnerability coverage — properties increasingly demanded of explainable, glass-box intelligent software systems [13].

7.1 Inconclusive Verdicts

The paper's results treat *Inconclusive* as a negative prediction; under a triage-oriented convention these cases are instead surfaced as deferred findings for analyst review. On the CWE-Bench-Java test holdout this changes little: precision is unchanged at 0.77 and F1 rises marginally from 0.76 to 0.77, so the main effect of *Inconclusive* handling is whether ambiguous cases are counted as misses or deferred for review. The overall Inconclusive Verdict Rate is 11% (8% on CWE-Bench-Java, 14% on CVEfixes), consistent with cleaner labels and more easily recoverable evidence in CWE-Bench-Java, whereas authorization bypass and SSRF remain harder to resolve from static evidence alone. Operationally, the important point is that GraphLedger makes this uncertainty explicit instead of forcing a binary decision.

7.2 Practical Limits

The current experiments use bounded verification with ℓ=20 and m=8. These values cover the inter-procedural depths seen in the evaluated repositories, but deeper paths can still be missed; in those cases the hypothesis remains *Inconclusive*. Preliminary validation on CWE-Bench-Java suggested that smaller bounds reduce recall, while larger bounds increase runtime with little added benefit.

The evaluation should also be read within scope. GraphLedger is best suited to taint-style vulnerabilities with identifiable source–sink structure, and the experiments focus on CWE-Bench-Java and CVEfixes. It does not yet cover logic flaws, timing-based vulnerabilities, or configuration errors, and final detection quality still depends on hypothesis generation quality.

7.3 Residual Errors and Future Work

Most remaining errors arise from limits in the underlying representation rather than instability in the ledger itself. Aliasing imprecision, incomplete CPG extraction, conservative dynamic-dispatch handling, and framework-specific callbacks can all distort source–sink evidence. This is especially visible in framework-heavy deserialization and SSRF cases, where static recovery remains difficult.

Failure Case Study. A representative false negative is DSpace (CVE-2022-31195, CWE-22): the LLM generated a correct hypothesis but bounded evidence retrieval resolved none of its assumptions, yielding *Inconclusive*. A second instance in the same repository was *Verified*, showing that the margin between hit and miss is evidence reachability, not hypothesis quality. Per-case ledger traces and a fuller failure breakdown are in supplementary Appendix J.

Two additional limitations: behavioral summaries are heuristic rather than formally verified (*TaintPres* = 0.88; Section 6.2), and the rule-based evidence classifier is not adversarially robust. Future work should focus on call-graph precision, framework modeling, adaptive search bounds, and robustness against adversarial evidence.

8 Related Work

Repository-scale vulnerability detection draws on four lines of work: static analysis, graph-based program representation, context compression, and LLM-assisted reasoning. Traditional static analyzers such as CodeQL and Infer [7, 19] provide broad coverage, but they often pay for that coverage with false positives because conservative approximations preserve infeasible paths. More generally, inter-procedural taint analysis and demand-driven data-flow analysis confront the same precision–scalability trade-off through propagation schemes, specification inference, and selective refinement of alias and path information. Code Property Graphs and related dependency graphs strengthen this line of work by unifying syntax, control flow, and data dependence in a single representation [5, 26]. Graph-learning approaches such as HGIVul [22], VulPatrol [9], and HAGNN [18] also exploit this structure, but depend on learned representations rather than explicit evidence-based verification.

More recent work combines graph-based analysis with LLMs. IRIS [16] improves coverage by inferring missing taint specifications for custom APIs, while SemTaint [6] uses multi-agent reasoning to recover missed dynamic call edges. A related line of work focuses on context selection at repository scale. Generic prompt-compression methods such as LLMLingua [12] are not designed for program analysis, because token-level reduction can disrupt syntax and dependency structure. By contrast, LongCodeZip [21], LLMxCPG [14], SnapVuln [25], VulTrigger [17], and VulnSC [24]

preserve vulnerability-relevant structure through slicing or behavioral abstraction. Our AACC component is closest to this second group: its goal is not generic prompt shortening, but preserving the dependency structure needed for downstream reasoning.

Another nearby line of work treats vulnerability detection as an iterative agentic process over repository context, an emerging direction whose trends and open challenges are surveyed in [10]. SWE-agent [28] and AutoCodeRover [30] show the broader utility of tool-using agents in software engineering, while LLMAO [27] demonstrates that LLMs can localize faults directly from program semantics. In security analysis, RepoAudit [8] performs repository auditing through graph traversal and memory-backed reasoning, and ClearAgent [2] applies agentic validation to binary analysis. VulAgent [23] is especially close to our setting: it generates candidate vulnerability hypotheses and then validates them through a dedicated agent. The main difference is that its validation quality still depends heavily on the fidelity of the provided context.

GraphLedger couples compression and verification more tightly than prior work. Relative to LLMxCPG, AACC uses a bidirectional source–sink corridor with retention predicates preserving ledger-query evidence; relative to VulAgent, it validates hypotheses over compressed representations with empirical path preservation (§3); relative to SnapVuln [25] and VulnSC [24], compression is co-designed with downstream verification rather than applied as a separate step. This integrated design improves token reduction and context efficiency over LLMxCPG under matched conditions (Table 7). Comparisons should be read under a shared evaluation regime rather than as claims about every baseline's best native setup.

9 Conclusion

We formulated repository-level vulnerability detection as structured hypothesis verification over compressed dependency graphs and introduced GraphLedger, which couples AACC with an Assumption Ledger for evidence-driven validation.

Across CWE-Bench-Java and CVEfixes, GraphLedger reduces false discoveries while maintaining competitive recall. The results support the central claim: repository-scale reasoning is more reliable when compression preserves vulnerability-relevant structure and decisions are constrained by explicit assumption checks. Gains are strongest in precision, FDR reduction, and auditability rather than coverage. Current limits include bounded search, Java-centered evaluation, and dependence on hypothesis quality. Future work should extend the method to broader languages and frameworks while preserving the same verification-oriented design.

10 Ethical Considerations

This work raises no direct ethical concerns and complies with the ACM Code of Ethics. It uses only public, already-disclosed vulnerability benchmarks (CWE-Bench-Java [11, 14], CVEfixes [1]), with no human subjects, private data, or zero-day disclosure. As a defensive tool, GraphLedger aims to help developers find and remediate vulnerabilities; its findings should be validated before action, which is exactly the auditable verification it advocates.

Data Availability Statement

The replication package is archived on Zenodo (DOI: 10.5281/zenodo.21356624) and submitted to the ASE 2026 Artifact Evaluation track. It contains the source code, prompts, the frozen `seed=42` validation/test splits, per-case prediction and assumption-ledger logs, evaluation scripts, the extended appendices (supplementary Appendices A–K), and a pre-built Docker image. All reported metrics can be recomputed offline from the archived predictions; only live re-runs need an OpenAI API key. The original hyperparameter-tuning validation split was not preserved, so the archive ships the documented `seed=42` split (§5) with its sensitivity analyses.

References

[1] Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. Association for Computing Machinery, New York, NY, USA, 30–39. doi:10.1145/3475960.3475985

[2] Xiang Chen, Anshunkang Zhou, Chengfeng Ye, and Charles Zhang. 2025. ClearAgent: Agentic Binary Analysis for Effective Vulnerability Detection. In *ACM SIGPLAN Workshop on Language Models and Programming Languages (LMPL)*. Association for Computing Machinery, New York, NY, USA, 12 pages. doi:10.1145/3759425.3763397

[3] Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2026. Vul-RAG: Enhancing LLM-Based Vulnerability Detection via Knowledge-Level RAG. *ACM Transactions on Software Engineering and Methodology* to appear, to appear (2026), 30 pages. doi:10.1145/3797277 Just Accepted.

[4] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Ankit Mody, Sarah Truitt, and Jonathan Larson. 2024. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. arXiv:2404.16130 [cs.CL] doi:10.48550/arXiv.2404.16130

[5] Mafalda Ferreira, Miguel Monteiro, Tiago Brito, Miguel E. Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. 2024. Efficient Static Vulnerability Analysis for JavaScript with Multiversion Dependency Graphs. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 164 (2024), 25 pages. doi:10.1145/3656394

[6] Jonah Ghebremichael, Saastha Vasan, Saad Ullah, Greg Tystahl, David Adei, Christopher Kruegel, Giovanni Vigna, William Enck, and Alexandros Kapravelos. 2026. Multi-Agent Taint Specification Extraction for Vulnerability Detection. arXiv preprint arXiv:2601.10865. arXiv:2601.10865 doi:10.48550/arXiv.2601.10865

[7] GitHub. 2026. CodeQL. https://codeql.github.com/. Accessed: 2026-02-25.

[8] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. 2025. RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, Vancouver, Canada, 21083–21100. https://proceedings.mlr.press/v267/guo25n.html

[9] Asmaa Hailane, Paria Shirani, and Guy-Vincent Jourdan. 2025. VulPatrol: Interprocedural Vulnerability Detection and Localization through Semantic Graph Learning. In *Proceedings of the 15th ACM Conference on Data and Application Security and Privacy (CODASPY)*. Association for Computing Machinery, New York, NY, USA, 401–412. doi:10.1145/3714393.3726509

[10] Hoang-Quoc-Bao Hua and Xuan-Bach Le. 2026. Emerging Trends and Challenges Toward LLM Agents in Static Analysis. In *Proceedings of the 39th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE 2026)*. Springer, Kuala Lumpur, Malaysia, to appear pages.

[11] IRIS SAST. 2025. CWE-Bench-Java: A manually vetted dataset for security vulnerability detection in Java projects. https://github.com/iris-sast/cwe-bench-java. GitHub repository, accessed 2026-03-25.

[12] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Singapore, 13358–13376. doi:10.18653/v1/2023.emnlp-main.825

[13] Thi-Hong-Cuc Le and Xuan-Bach Le. 2027. From Black-Box to Glass-Box: Explainable AI for Applied Intelligent Software Systems Across the Software Development Life Cycle. In *Advances and Trends in Artificial Intelligence. Theory and Applications*, Hamido Fujita, Ali Selamat, Masitah Ghazali, and Moonis Ali (Eds.). Springer Nature Singapore, Singapore, 88–105. doi:10.1007/978-981-92-2891-1_8

[14] Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models. In *Proceedings of the 34th USENIX Security Symposium*. USENIX Association, Seattle, WA, USA, 489–507.

[15] Yikun Li, Ting Zhang, Jieke Shi, Chengran Yang, Junda He, Xin Zhou, Jinfeng Jiang, Huihui Huang, Wen Bin Leow, Yide Yin, et al. 2026. Beyond Function-Level Analysis: Context-Aware Reasoning for Inter-Procedural Vulnerability Detection. arXiv preprint arXiv:2602.06751. arXiv:2602.06751 doi:10.48550/arXiv.2602.06751

[16] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. arXiv preprint arXiv:2405.17238. arXiv:2405.17238 doi:10.48550/arXiv.2405.17238

[17] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the Effectiveness of Function-Level Vulnerability Detectors for Inter-Procedural Vulnerabilities. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, New York, NY, USA, 12 pages. doi:10.1145/3597503.3639218

[18] Yu Luo, Weifeng Xu, and Dianxiang Xu. 2025. Detecting Code Vulnerabilities with Heterogeneous GNN Training. arXiv preprint arXiv:2502.16835. arXiv:2502.16835 doi:10.1007/s10207-025-01132-x

[19] Meta. 2026. Infer Static Analyzer. https://fbinfer.com/. Accessed: 2026-02-25.

[20] OpenAI. 2024. GPT-4o System Card. https://openai.com/research/gpt-4o-system-card. Accessed: 2026.

[21] Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2025. LongCodeZip: Compress Long Context for Code Language Models. arXiv preprint arXiv:2510.00446. arXiv:2510.00446 doi:10.1109/ASE63991.2025.00020

[22] Zihua Song, Junfeng Wang, Kaiyuan Yang, and Jigang Wang. 2023. HGIVul: Detecting inter-procedural vulnerabilities based on hypergraph convolution. *Information and Software Technology* 160 (2023), 107219. doi:10.1016/j.infsof.2023.107219

[23] Ziliang Wang, Ge Li, Jia Li, Hao Zhu, and Zhi Jin. 2025. VulAgent: Hypothesis-Validation Based Multi-Agent Vulnerability Detection. arXiv preprint arXiv:2509.11523. arXiv:2509.11523 doi:10.18653/v1/2026.findings-acl.928

[24] Bozhi Wu, Chengjie Liu, Zhiming Li, Yushi Cao, Jun Sun, and Shang-Wei Lin. 2025. Enhancing Vulnerability Detection via Inter-Procedural Semantic Completion. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 22 pages. doi:10.1145/3728912

[25] Bozhi Wu, Shangqing Liu, Yang Xiao, Zhiming Li, Jun Sun, and Shang-Wei Lin. 2023. Learning Program Semantics for Vulnerability Detection via Vulnerability-Specific Inter-Procedural Slicing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Association for Computing Machinery, New York, NY, USA, 1371–1383. doi:10.1145/3611643.3616351

[26] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE, San Jose, CA, USA, 590–604. doi:10.1109/SP.2014.44

[27] Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent J. Hellendoorn. 2024. Large Language Models for Test-Free Fault Localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)* (Lisbon, Portugal). Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. doi:10.1145/3597503.3623342

[28] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv preprint arXiv:2405.15793. arXiv:2405.15793 doi:10.52202/079017-1601

[29] Bing Zhang, Xu Zhi, Meng Wang, Rong Ren, and Jun Dong. 2025. Enhancing Java Web Application Security: Injection Vulnerability Detection via Interprocedural Analysis and Deep Learning. *IEEE Transactions on Reliability* 74, 3 (2025), 3642–3656. doi:10.1109/TR.2024.3521381

[30] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, New York, NY, USA, 1592–1604. doi:10.1145/3650212.3680384